Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2012) 2197–2205

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

Implementing Intelligent Cores using Processor Virtualization for Fault Tolerance

Blesson Varghese[1,*], Gerard McKee[2,*], Vassil Alexandrov[3,*]

School of Systems Engineering, University of Reading, Whiteknights Campus
Reading, Berkshire, United Kingdom, RG6 6AY

Abstract

Processor virtualization for process migration in distributed parallel computing systems has formed a significant component of research on load balancing. In contrast, the potential of processor virtualization for fault tolerance has been addressed minimally. The work reported in this paper is motivated towards extending concepts of processor virtualization towards 'intelligent cores' as a means to achieve fault tolerance in distributed parallel computing systems. Intelligent cores are an abstraction of the hardware processing cores, with the incorporation of cognitive capabilities, on which parallel tasks can be executed and migrated. When a processing core executing a task is predicted to fail the task being executed is proactively transferred onto another core. A parallel reduction algorithm incorporating concepts of intelligent cores is implemented on a computer cluster using Adaptive MPI and Charm++. Preliminary results confirm the feasibility of the approach.

Keywords: fault tolerance, intelligent cores, processor virtualization, adaptive MPI, parallel reduction

1. Introduction

Process migration is an important concept in homogeneous or heterogeneous distributed systems irrespective of where and how computing nodes are situated geographically. The concept of process migration investigates how a process can be transferred from one computing node to another and lays emphasis on the mobility of a process [1].

Process migration models can be classified on the basis of the type of state information required for migration and level of migration support implementation.

Three types of migration models are considered in [2] based on the type of state information required for process migration. Firstly, the minimal state migration model that requires minimum information for simple process migration. Secondly, the full state migration model that facilitates migration of complex process involving the network, files and

*Corresponding author

Email addresses: b.varghese@student.reading.ac.uk (Blesson Varghese), g.t.mckee@reading.ac.uk (Gerard McKee), v.n.alexandrov@reading.ac.uk (Vassil Alexandrov)

[1]Blesson Varghese is a PhD candidate at the Active Robotics Laboratory, School of Systems Engineering, University of Reading

[2]Dr. Gerard McKee is Senior Lecturer in Networked Robotics at the School of Systems Engineering, University of Reading

[3]Prof. Vassil Alexandrov is Director of the Centre for Advanced Computing and Emerging Technologies (ACET), University of Reading

1877-0509
doi:10.1016/j.procs.2010.04.246

threading. Thirdly, the distributed state migration model that requires large amount of state information concerning process distributed over several computing nodes.

Three types of migration models are considered in [3] based on the level of migration support implementation. Firstly, the kernel-level migration model in which migration support is implemented in the operating system and supports preemptive migrations. Such kernel-level models tend to be complex. Secondly, the user-level migration model in which migration support is implemented as libraries in user space and linked to user applications at compile time. Thirdly, the application-level migration model in which migration support is implemented within the application itself and thereby increases potential for heterogeneous computations.

However, whatever be the migration model implemented for achieving process migration, there are three phases involved in any process migration strategy [4]. Firstly, the detaching phase that isolates the process to be migrated from the current computing environment. Secondly, the transfer phase that shifts the process to a target node. Thirdly, the attaching phase that reinstates the process execution by making available all required resources on the target node.

There are two enabling technologies that facilitate process migration. Firstly, by checkpointing in which the state of a process requiring migration is saved on disk and then restarted on a node using the saved state information. In [3], [5] and [6] checkpointing based process migration is reported.

However, checkpointing, a traditional approach for process migration is challenged by a few drawbacks that pose constraints on achieving effective process migration. Firstly, checkpointing lacks the flexibility to cope with dynamic changes in the environment [1]. Secondly, checkpointing requires server based coordination thereby increasing overheads involved in migration and such strategies are susceptible to single point failures [7]. Thirdly, checkpointing is associated with a cold restart from the nearest checkpoint thereby increasing execution time of a process [8].

Secondly, by processor virtualization a more complex strategy that abstracts processors into virtual processors and enables migration of processes being executed on a virtual process. Processor virtualization is more advantageous since the challenges in checkpointing based process migration are overcome and tend to be scalable in heterogeneous environments. Processor virtualization based migration is reported in [9] and [10].

To implement process migration technologies, middleware layers are chosen for the purpose and are reported in research. Research based on process migration technologies implemented in middleware layers is reported in [1], [3], [11], [12] and [13].

Most research reported above considers process migration for load balancing and scheduling in distributed systems. The potential of process migration by processor virtualization for achieving fault tolerance in distributed computing systems does not seem to have been the focus of most research undertaken in the area. Though few research work reported in [14], [15] and [16] is based on processor virtualization fault tolerance, practical usage is sparsely reported. Hence there is a necessity to amply exploit its potential.

The work reported in this paper is motivated towards extending the idea of processor virtualization for implementing 'intelligent cores' on a computer cluster for achieving fault tolerance. The approach aims to implement intelligent cores as autonomous swarm agents in an arena on which parallel tasks can be executed and migrated. The intelligent cores interact with each other to transfer executing tasks from one processing core to another during the event of a hardware failure.

The remainder of this paper is organized as follows. Section 2 presents concepts of processor virtualization useful for the intelligent core approach. Section 3 identifies parallel reduction algorithm that would benefit from the intelligent core approach. Section 4 deals with the implementation details of the intelligent core approach. Section 5 presents results and a discussion on the implementation. Section 6 concludes the paper by considering future work.

2. Intelligent Cores

In the 'Intelligent Core' approach proposed in this paper, the emphasis is laid towards achieving fault tolerance in parallel computing systems. In abstract terms the approach can be summarized as follows. A parallel task to be executed resides within a queue and is scheduled onto available cores for processing of a parallel computing system. In an ideal scenario, the task scheduled onto the parallel computing system is executed without any failure. However, in real-time computations subject to failures and errors, the processing cores need to interact with one another to transfer tasks from one core to another when a hardware failure is predicted to occur. Figure 1 (left) illustrates the intelligent core approach.

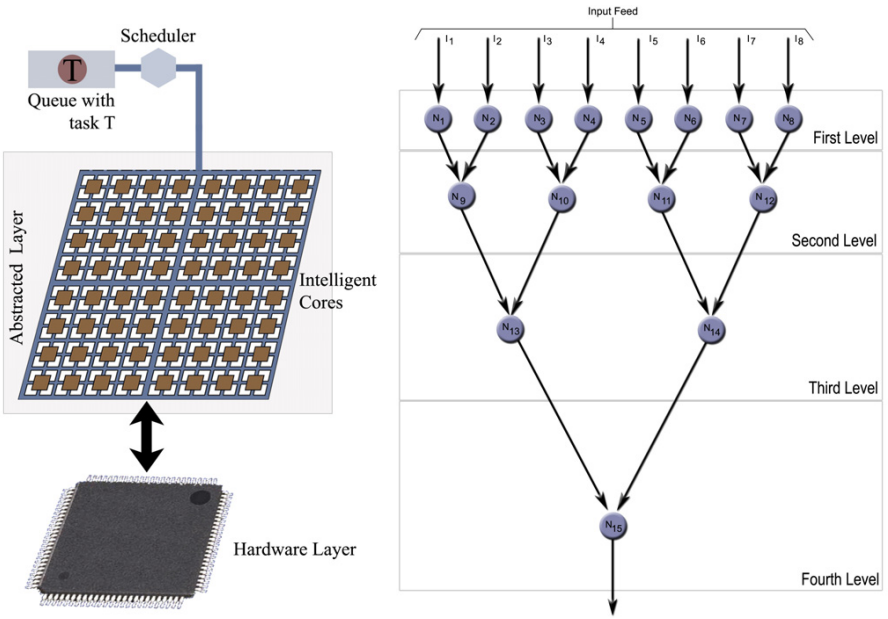

Figure 1: Illustration of the Intelligent Core Approach (left) and illustration of the parallel summation algorithm to be executed on the intelligent cores (right)

The intelligent core approach differs from intelligent agent approaches for fault tolerance reported in [17] in three different ways. Firstly, in the intelligent core approach intelligence is embodied within an abstracted core unlike the intelligent agent approach in which an agent executing a task is intelligent. Secondly, in the intelligent core approach, a processing core is responsible for the transfer of a task when a failure is predicted to occur as against the intelligent agent approach in which an executing agent transfers from a core predicted to fail to another core. Thirdly, in the intelligent core approach agent-process and process-agent communication of hardware sensory information is eliminated unlike the intelligent agent approach in which a hardware probing process on a core communicates with task carrying agents on the core.

To achieve intelligence on processing cores so as to achieve fault tolerance there are two fundamental requirements. Firstly, an abstraction over hardware cores so as to embody intelligence. This abstraction provides a virtual arena onto which tasks can be scheduled. Intelligent cores are hence required to be implemented as autonomous agents seen in natural swarms and form the virtual arena referred to as the landscape that represents the computing space.

Secondly, cognitive capabilities of the abstracted cores so as to demonstrate intelligence as seen in natural swarm agents. Four necessary cognitive capabilities of the abstracted cores required to achieve fault tolerance are identified. Firstly, a core is capable of being able to know its environment, the surroundings in which it is located. Secondly, a core is capable to identify other cores in its vicinity on which a task can nicely situate. Thirdly, a core is capable to sense any hazard that is likely to deteriorate or impair the execution of a task on the core. Fourthly, a core is capable to transfer a task executing on it to another core when necessary for survival.

The intelligent core approach bridges together the two requirements considered above. The abstraction layer is capable of anticipating core failures and in effect react to the anticipated situation so as to continue execution, without the necessity of a process restart, and achieve fault tolerance. If a processing core fails, the process which was executed on the core needs to be moved to another core where resources previously accessed can be utilized. Once a process has been moved or relocated, all data dependencies also need to be re-established.

3. Parallel Reduction Algorithms on Intelligent Cores

The nature of tasks or algorithms that would benefit from the intelligent core approach is considered in this section. Parallel reduction algorithms [18], which implement the bottom-up approach of binary trees, are of interest in the context of the intelligent core approach due to two main reasons. Firstly, the computing nodes of a parallel reduction algorithm tend to be critical. The execution of the algorithm stalls or produces an incorrect solution if any node information is lost. Secondly, parallel reduction algorithms are employed in critical applications such as space applications. These applications require fault tolerant and self-managing distributed systems. Parallel summation is an exemplar of parallel reduction algorithm considered in this paper.

Figure 1 (right) illustrates the parallel reduction algorithm considered in this paper. The algorithm works in four sequential levels. The first level comprising nodes $N_1 - N_8$ receives a live input feed of data. The second level comprising nodes $N_9 - N_{12}$ receives data from the first level, adds the data received and yields the result to the third level nodes N_{13} and N_{14}. The fourth level, adds data received from the third level nodes and produces the final result.

For a given time step, every node in a level operates in parallel. Each node is characterized by input dependencies (process or processor a node is dependent on for receiving an input), output dependencies (process or processor a node yields data to as output) and data contained in the node. The first level nodes have one input dependency and one output dependency. For instance, node N_1 has one input dependency I_1 and node N_9 as its output dependency. However, the second, third and fourth levels have two input dependencies and one output dependency. For instance, node N_{13} of the third level has nodes N_9 and N_{10} as input dependencies and node N_{15} as output dependency. The data contained in a node is either the input data for the first level nodes or a calculated value (sum of two values in the case of a parallel summation algorithm) stored within a node.

However, since the classic version of parallel reduction algorithm does not incorporate fault tolerant concepts, a fault tolerant version adhering to the concepts of the intelligent core approach is implemented.

4. Implementing Intelligent Cores for Fault Tolerance

The implementation of the parallel reduction algorithm incorporating fault tolerant concepts using intelligent cores is considered in this section. The resources and middleware employed in the implementation are reported.

The cluster used for the research reported in this paper is one among the high performance computing resources available at the Centre for Advanced Computing and Emerging Technologies (ACET), University of Reading, United Kingdom [19][20]. The cluster is primarily used for the purpose of teaching and performing multi-disciplinary research. The cluster consists of a head node and 33 compute nodes. The formal specification of the head node is an Intel Pentium 4 CPU 3.20 GHz, 2 GB RAM and 160 GB hard disk, while that of 31 compute nodes are Intel Pentium 4 CPU 2.40 GHz, 512 MB RAM and 80 GB hard disk, and that of the remaining 2 compute nodes are Intel Pentium 4 CPU 2.60 GHz, 512 MB RAM and 40 GB hard disk. All nodes are connected via a Gigabit Ethernet switch and communicate via the standard TCP protocol.

To implement the intelligent core approach Adaptive MPI (AMPI) [15] is used as the necessary middleware developed at the Parallel Programming Laboratory, University of Illinois at Urbana-Champaign [21]. AMPI is an implementation of MPI 1.1 [22][23] standards developed over Charm++[24], a C++ based parallel programming language. The aim of AMPI is to achieve dynamic load balancing by utilizing objects capable of migration in CHARM++.

The parallel reduction algorithm implemented in AMPI would be similar to an implementation in traditional MPI. However, a few changes are required in traditional MPI programs for efficient execution using AMPI. Privatization of global variables used in MPI programs is the first necessity. This is possible by using the -swapglobals flags during compile and link time or by manually changing the structure of the program. The AMPI developer instructions for the privatization of global variables are reported in [25].

Another concept associated with AMPI is chunks and chunkification [25]. When a traditional MPI program is converted to an AMPI program, subsequently an MPI process is converted into a chunk, a combination of a user-level thread and the data it manipulates. This conversion is referred to by the AMPI developers as chunkification.

The mapping of virtual processors to physical processors in AMPI is specified as a run time option [25]. The predefined block mapping scheme that maps virtual processors to physical processors in chunks is used for running the implemented program reported in this paper. The BLOCK_MAP flag is used at runtime. For example, if four virtual processors are mapped onto two physical processors using the block mapping scheme, then virtual processors

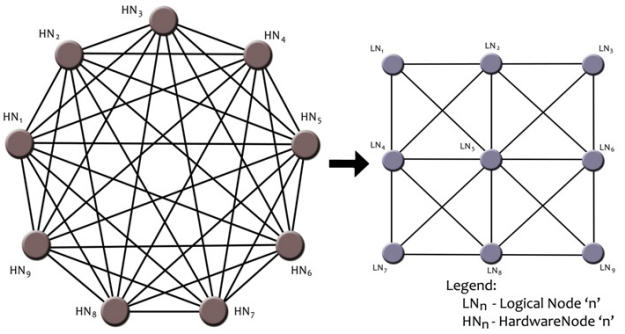

Figure 2: Mapping hardware nodes to logical nodes

identified as 0 and 1 is mapped to physical processor 0 and virtual processors identified as 2 and 3 are mapped to physical processor 1.

As considered in section 4, the fault tolerant parallel reduction algorithm employing the concepts of the intelligent core approach also employs four sequential levels, with the same nodes and configuration. The nodes in this case are also characterized by input dependencies, output dependencies and data contained in the node, but are implemented on the abstracted hardware layer considered later in this section.

In the intelligent core approach, the major constituent that comes to play is the abstraction of hardware resources by processor virtualization. In other words, the intelligent cores are an abstract view of the hardware cores. The hardware layer comprising the physical nodes of the cluster is connected via a switch, thereby forming a fully connected mesh topology. However, the abstraction is obtained when the physical nodes are abstracted as logical nodes. This is possible by implementing rules/policies. The policies are such that an abstracted core can only communicate with a vertically, horizontally or diagonally adjacent core, effectively leading to a grid topology on the abstracted layer. For example, nine nodes forming a fully connected mesh topology in figure 2 is abstracted to a grid topology in the abstraction layer.

Intelligence of the core is embodied in the abstraction layer such that intelligence is demonstrated cognitively in four different ways. Firstly, a core updates knowledge of its surrounding by monitoring local neighbours. Independent of what the cores are executing, the cores can monitor each other. Each core can ask the question 'Are you alive' off its neighbours and gain information.

Secondly, the constant update of information on a cores locality enables the core to know which neighbour nodes are capable to execute a task if necessary.

Thirdly, a core constantly monitors itself and predicts if a failure is likely to occur on it. In the implementation reported in this paper sensory information on whether a core is likely to fail is based on simulating core temperature. When the temperature of a core rises beyond a threshold, the core predicts that it is likely to fail, on similar lines to proactive fault tolerance.

Fourthly, if a core is likely to fail, it must be capable to transfer a task executing on it to another core and adjust to failure. Once a process has relocated all data dependencies need to be reestablished. AMPI provides a useful subroutine MPI_Migrateto that enables the transfer of processes across virtual processors.

The MPI_Migrateto subroutine facilitates the transfer of a process executing on a core in the abstracted layer to another core. This subroutine automatically reinstates the necessary dependencies. Thus, when a failure is predicted data for summation either obtained from a previous level node or a calculated value to be yielded to a next level, is not lost and does not affect the final solution if employed in a critical application.

Two scenarios based on single node failure (only one node fails in an instant) were considered in the intelligent core approach during implementation. Firstly, a scenario that assumed no cores connected to a core predicted to fail in the abstracted layer would fail in a consecutive time step. Figure 3 illustrates the sequence of events in the first scenario.

Secondly, a scenario that was more realistic in nature and assumed that any core connected to a core predicted to fail in the abstracted could fail in a consecutive time step. Figure 4 illustrates the sequence of events in the second

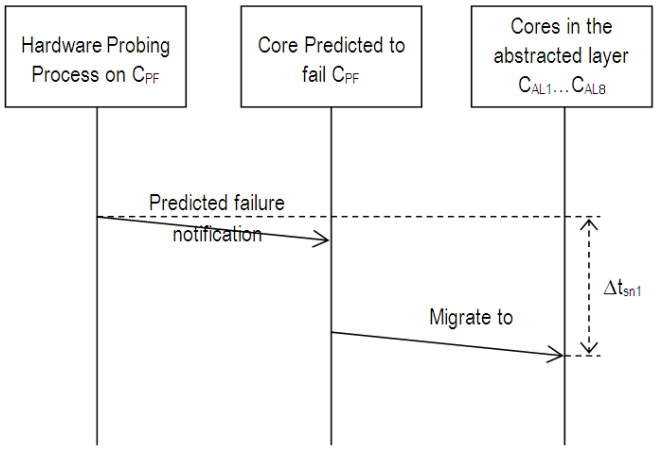

Figure 3: Communication sequence for the first single node failure scenario

scenario. Additional communication between the core predicted to fail (C_{PF}) and the hardware probing process ($P_{C_{AL1}}...P_{C_{AL8}}$) of the cores connected in the abstracted layer to core C_{PF} is employed. The additional communication sequence is necessary so as to select a target core, from among the eight connected cores $C_{AL1} - C_{AL8}$ in the abstracted layer, onto which the process from C_{PF} can be migrated.

In short, though the fault tolerant version of the parallel reduction algorithm reported in this paper only considers single point failures, an implementation incorporating concepts of the intelligent core approach is obtained. Preliminary results gathered from the experimental setup are considered in the next section.

5. Results

The mean time taken to transfer a task executing on a core predicted to fail onto another core in the abstracted layer was calculated for 30 independent experimental runs executing the parallel reduction algorithm. Figure 5 and figure 6 are graphs plotted on the basis of the two scenarios assuming only single node failures considered in the previous section. The data set for plotting the graphs were obtained during experimental runs by noting the time taken for task migration from computational node N_{14} during each run to another node in the abstracted layer.

The mean time calculated in the first scenario is obtained as 0.263 sec and is indicated in red in figure 5. The mean time calculated in the second scenario is obtained as 0.294 *sec* and is indicated in red in figure 6. In other words, the mean of $\Delta t_{sn1} = 0.263$ *sec* and $\Delta t_{sn2} = 0.294$ *sec*.

The mean time of the second scenario, a more realistic scenario, tends to be greater due to the additional communication between the core predicted to fail (C_{PF}) and the hardware probing process ($P_{C_{AL1}}...P_{C_{AL8}}$) of the cores connected in the abstracted layer to core C_{PF}. This additional time referred to as Δt_x, in figure 3 (bottom) is obtained as $\Delta t_{sn2} - \Delta t_{sn1} = 0.031$ *sec*

The significance of the mean time of $\Delta t_{sn2} = 0.294$ *sec* for proactively migrating a task in a realistic scenario is apparent from the fact that the order of the calculated value is in milliseconds. This implies that the time taken for reinstating execution in the intelligent core approach proposed in this paper is much less when compared to the time taken in traditional fault tolerant techniques like checkpointing that require cold restarts and occasional intervention by the administrator.

Though the results presented in this paper are preliminary, they confirm the feasibility of the proposed intelligent core approach and its pertinence towards fault tolerance.

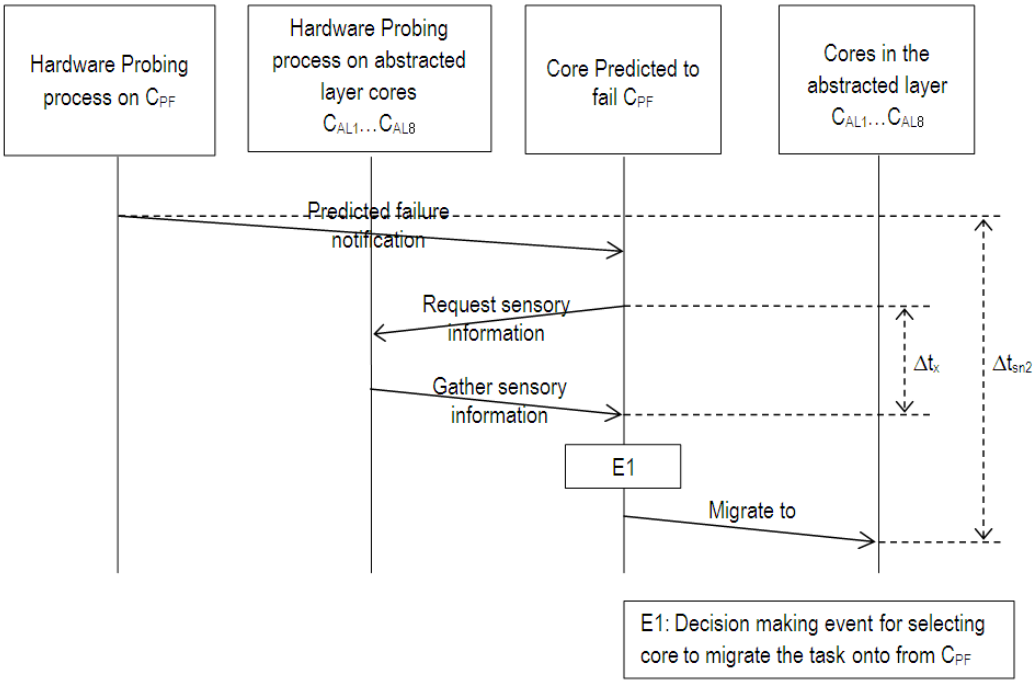

Figure 4: Communication sequence for the second single node failure scenario

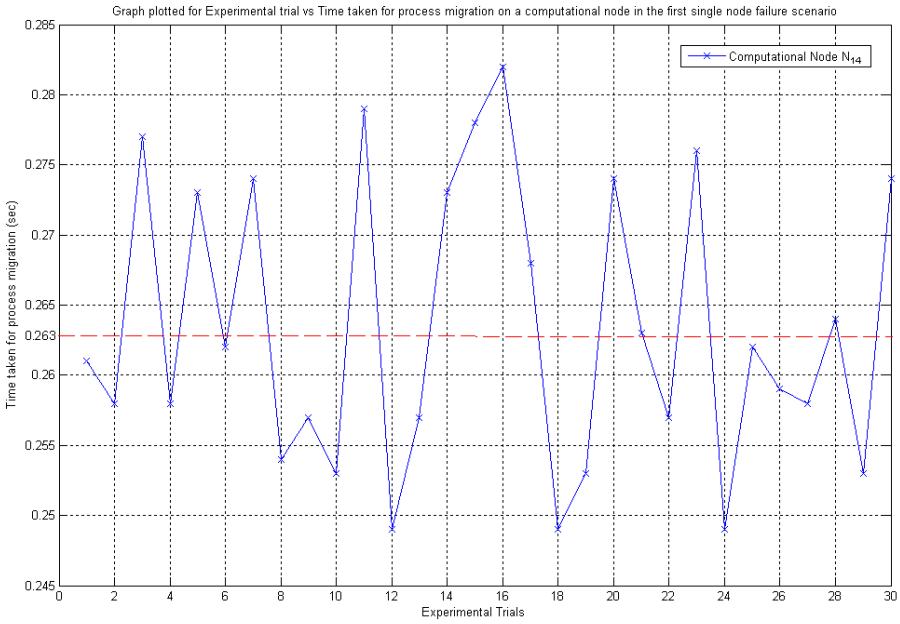

Figure 5: Graph plotted for experimental trials vs. time taken for process migration for the first single node failure scenario

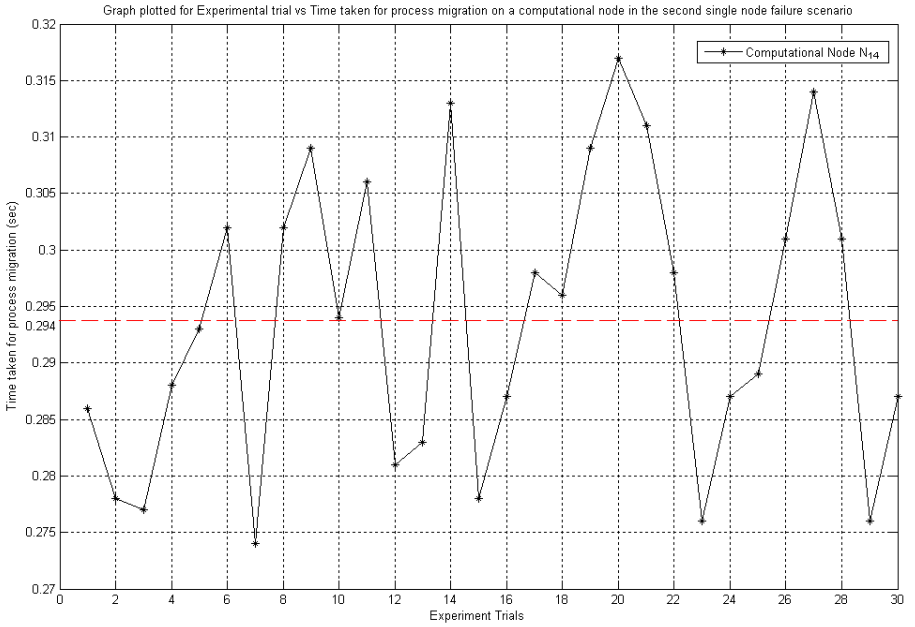

Figure 6: Graph plotted for experimental trials vs. time taken for process migration for the second single node failure scenario

6. Conclusion

In this paper, the use of processor migration by processor virtualization for fault tolerance is presented. The purpose of processor virtualization is extended towards 'intelligent cores', an abstraction over processing cores that contributes towards intelligent core behaviour. The proposed approach considers the migration of a task being executed on a core predicted to fail onto another core in the abstracted layer. The proposed approach is implemented on a cluster for parallel reduction algorithms employing Adaptive MPI and Charm++ as the appropriate middleware. Preliminary results confirm the feasibility of the proposed and are a motivation to further investigate the approach.

Future work will aim to gather more statistics from experiments based on the intelligent core approach and compare it with the intelligent agent approach reported in [17]. The feasibility of the approach on other parallel distributed computing systems will be considered. Immediate efforts will be made to implement support for multiple core failures occurring simultaneously and explore the possibility of implementing the intelligent core approach on other existing middleware.

References

[1] R. S. C. Ho, C. -L. Wang and F. C. M. Lau, "Lightweight Process Migration and Memory Prefetching in openMosix" in the Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 2008, pp. 1–12.

[2] T. Boyd and P. Dasgupta, "Process Migration: A Gen eralized Approach using a Virtualizing Operating System" in the Proceedings of the 22nd International Conference on Distributed Computing Systems, 2002, pp. 385–392.

[3] H. Jiang and V. Chaudhary, "Process/Thread Migration and Checkpointing in Heterogeneous Distributed Systems" in the Proceedings of the 37th Hawaii International Conference on System Sciences, 2004.

[4] I. Zoraja, A. Bode and V. Sunderam, "A Framework for Process Migration in Software DSM Environments" in the Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing, 2000, pp. 158–165 .

[5] G. Vallee, C. Morin, J. -Y. Berthou, I. D. Malen and R. Lottiaux, "Process Migration based on Gobelins Distributed Shared Memory" in the Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002.

[6] J. Cao, Y. Li and M. Guo, "Process Migration for MPI Applications based on Coordinated Checkpoint" in the Proceedings of the 11th International Conference on Parallel and Distributed Systems, 2005, pp. 306–312.

[7] J. P. Walters and V. Chaudhary, "Replication-Based Fault Tolerance for MPI Applications" in the IEEE Transactions on Parallel and Distributed Systems, Vol. 20, No. 7, July 2009, pp. 997–1010.
[8] X. Yang, Y. Du, P. Wang, H. Fu and J. Jia, "FTPA: Supporting Fault-Tolerant Parallel Computing through Parallel Recomputing" in the IEEE Transactions on Parallel and Distributed Systems, Vol. 20, Issue 10, October 2009, pp. 1471–1486.
[9] L. V. Kale, "Performance and Productivity in Parallel Programming via Processor Virtualization" in the Proceedings of the First International Workshop on Productivity and Performance in High-End Computing, 2004.
[10] S. Chakravorty and L. V. Kale, "A Fault Tolerance Protocol with Fast Fault Recovery" in the Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Symposium, 2007, pp. 1–10.
[11] R. K. K. Ma, C. -L. Wang and F. C. M. Lau, "M-JavaMPI: A Java-MPI Binding with Process Migration Support" in the Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002, pp. 255–262.
[12] Y. Zhang and J. Hu, "Checkpointing and Process Migration in Network Computing Environment" in the Proceedings of the International Conferences on Info-tech and Info-net, 2001, pp. 179–184.
[13] P. Czarnul and H. Krawczyk, "Parallel Program Execution with Process Migration" in the Proceedings of the International Conference on Parallel Computing in Electrical Engineering, 2000, pp. 50–54.
[14] C. Du and X. -H. Sun, "MPI-Mitten: Enabling Migration Technology in MPI" in the Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid, 2006, pp. 11–18.
[15] C. Huang, O. Lawlor and L. V. Kale, "Adaptive MPI" in the Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing, LNCS 2958, 2003, pp. 306–322.
[16] L. Kale, O. Lawlor and M. Bhandarkar, "Parallel Objects: Virtualization and in-Process Components" in the Proceedings of the Workshop on Performance Optimization for High-Level Languages and Libraries, 2002.
[17] B. Varghese, G.T. McKee and V. N. Alexandrov, "Intelligent Agents for Fault Tolerance: From Multi-Agent Simulation to Cluster-based Implementation" accepted for publication in the 3rd IEEE Workshop on Bio and Intelligent Computing, Perth, Australia, 2010.
[18] M. J. Quinn, "Parallel Computing Theory and Practice" McGraw-Hill, Inc. 1994.
[19] Center for Advanced Computing and Emerging Technologies (ACET) website: www.acet.reading.ac.uk
[20] High Performance Computing at ACET website: http://hpc.acet.rdg.ac.uk/
[21] Parallel Programming Laboratory website: http://charm.cs.uiuc.edu/
[22] MPI 1.1 Documents: http://www.mpi-forum.org/docs/docs.html
[23] MPI 1.1 Standards Index: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
[24] L. V. Kale and S. Krishnan, "CHARM++: Parallel Programming with Message-Driven Objects" in 'Parallel Programming using C++' (Eds. G. V. Wilson and P. Lu), pp. 175–213, MIT Press, 1996.
[25] Online Adaptive MPI Manual: http://charm.cs.uiuc.edu/manuals/html/ampi/manual.html